文法圧縮に基づいた圧縮データの自己索引構造化の提案 (計算機科学とアルゴリズムの数理的基礎とその応用)

AUTHOR(S):

馬場, 雅大; 丸山, 史郎; 坂本, 比呂志; 定兼, 邦彦; 山下, 雅史

CITATION:

馬場, 雅大 ...[et al]. 文法圧縮に基づいた圧縮データの自己索引構造化の提案 (計算機科学とアルゴリズムの数理的基礎とその応用). 数理解析研究所講究録 2011, 1744: 209-212

ISSUE DATE:

2011-06

URL:

http://hdl.handle.net/2433/170947

数理解析研究所講究録
第1744巻 2011年 209-212

2010年度冬のLAシンポジウム [S10]

文法圧縮に基づいた圧縮データの自己索引構造化の提案

馬場 雅大* 丸山 史郎† 坂本 比呂志‡ 定兼 邦彦§ 山下 雅史¶

1 はじめに

データを圧縮保存している場合，文字列検索や部分復元といった操作を行う際に全体を復元するのは不利益が大きい．それを回避する手段として自己索引 (self-index) 構造が提案されている．

本稿で扱うのは，文脈自由文法 (CFG) に基づいた圧縮と構文木を圧縮した全二分木の巡回によって文字列検索を行うものである [4]．このデータ構造は長さ u の文字列 S から変換された CFG G が持つ異なる規則 (変数) の数を n，G の構文木の高さを h としたとき，$1.5n\log n+n\log h+h\log n+o(n\log n)$ ビット領域で表し，長さ m のパタン P の出現回数を求める計算量は $O(m\log^2 n+occ_c(m\log n+h))$ である．CFG の最適解を n_* とすると $n=O(n_*\log u)$，$h=O(\log u)$ である．occ_c とは構文木中に現れるコアの出現回数を示す．コアとは P に含まれる部分文字列を十分に長く符号化した変数である．構文木で P が存在するところにはコアが存在するので検索時の必要条件とする．P が長い場合 occ_c の数は少なくなり検索時間は小さくなる．

一方でこの手法は，文字列検索以外の操作をサポートしていない．自己索引構造で効率よくサポートするべき操作を以下に定義する．

- $count(S,P)$: S 中の P の出現回数
- $locate(S,P)$: S 中の P の位置
- $access(S,i)$: S 中の i 番目の文字

*九州大学大学院システム情報科学府
†九州大学大学院システム情報科学研究院
‡九州工業大学大学院情報工学研究院
§国立情報学研究所
¶九州大学大学院システム情報科学研究院

2 準備

以下ではデータ構造で用いる簡潔データ構造などについて説明する．

簡潔データ構造は，対象となるデータ構造をそのデータの情報理論的下限に漸近的に一致する一致する簡潔表現と，データに対して何らかの操作を効率よく行い，漸近的に小さい簡潔索引とで成り立つ．情報理論的下限とは，位数 L に対して $\lg L$ ビットである．なお lg は底が 2 の対数を表している．長さ n のビット列であれば，$L=2^n$ 通りのパタンが考えられるので情報理論的下限は n ビットである．

2.1 Rank/select 演算

$|\mathcal{A}|=\sigma$ のアルファベット $\mathcal{A}$ 上の長さ n の文字列 S を考える．ここで S に対する操作を以下のように定義する．

- $rank_c(S,i)$: $S[..i]$ 中の c の出現回数
- $select_c(S,i)$: S 中で先頭から i 番目の c の位置
- $access(S,i)$: $S[i]$

このような操作を効率的にサポートするデータ構造を rank/select 辞書という．

2.1.1 ビット列の簡潔データ構造

長さ n のビット列のサイズは 1 の数が少なければ圧縮できる．1 の数が m 個のビット配列での操作を定数時間で行う $B(n,m)=\lceil\lg\binom{n}{m}\rceil+O(n\lg\lg n/\lg n)=m\lg\frac{n}{m}+\Theta(n)+O(n\lg\lg n/\lg n)$ ビットのデータ構造がある [6]．これを完全索引付辞書 (FID) と呼ぶ．

2.1.2 文字列の簡潔データ構造

アルファベットサイズが2よりも大きい場合にはウェーブレット木 [2] を用いる．これはテキスト S を表現する深さ $\lg\sigma$ の二分木であり，各ノードはビット列を格納している．ある深さにある列の長さを合計するとテキスト長 n となる．各列に対して rank/select 辞書を追加したサイズは $n\lg\sigma(1+o(1))$ ビットとなる．$rank_c$ などの操作の計算量は $O(\lg\sigma)$ である．

2.2 全二分木の簡潔データ構造

順序木の簡潔データ構造として BP 表現 [5] と DFUDS 表現 [1] が挙げられる．両者は全体としてバランスする開き括弧と閉じ括弧の括弧列で構成され，ノード数を n とすると長さは $2n$ である．

2.2.1 BP，DFUDS で共通するデータ構造

括弧列を用いて表現する BP，DFUDS では木の巡回を行うために，以下の操作を $o(n)$ ビットで定数時間で行う補助データ構造が提案されている [5]．対象となる括弧列を P とすると以下の通りである．

- $findopen(P,x)$: $P[x]$ にある開き括弧に対応する閉じ括弧の位置を返す．
- $findclose(P,x)$: $P[x]$ にある閉じ括弧に対応する開き括弧の位置を返す．
- $enclose(P,x)$: $P[x]$ にある括弧とそれに対応する括弧を囲う最小の括弧対の開き括弧の位置を返す．

これらと括弧列の rank/select 索引などにより木の巡回を実現する．

2.3 全二分木の簡潔データ構造

全二分木は情報理論的下限が $n-\Theta(\log n)$ ビットなので n ビットの別表現 [8] を用いる．

全二分木を前置順に巡回して内部ノードであれば開き括弧，葉ノードであれば閉じ括弧を順番に並べる．括弧全体をバランスするために先頭に開き括弧を配置する．この括弧列 F から以下のような操作を定数時間で行うことができる．なお x は前置順に並べたノードを表現しているとする．

- $isleaf(x)$: x は葉であるか．
- $parent/sibling(x)$: x の親・弟
- $left/rightchild(x)$: x の長男・次男
- $childrank(x)$: x がその親の左から何番目の子か
- $leaf\_rank/select$: 葉ノードの rank/select
- $inner\_rank/select$: 内部ノードの rank/select

全二分木の表現は，パトリシアトライや DFUDS 圧縮 [3] などが提案されている．主要項では同サイズであるが，パトリシアトライは右の子への対応に結局索引を追加する他，[3] は複雑であり，簡潔索引そのものは $2n$ ビットの括弧列に対するものなので括弧列そのものを n ビットとしている本稿の表現の方が有利である．

3 文法圧縮に基づく圧縮索引の自己索引構造化

本節では，edit sensitive parsing に基づいてテキスト S から構築された構文木 T に対する表現と操作について簡単に説明する．構文木とは $X\rightarrow AB$ といった生成規則を持つ CFG と等価な順序木のことを指す．このとき X から AB を求めるためのデータ構造を辞書 D とし，逆に AB から X を求める逆引き辞書を D^R と表記する．

パタン P を検索する際に，P の S における全ての出現に対して，P の部分文字列を十分に長く符号化した極大な変数 (規則) をコアとする．T 内に構文木が存在し，かつ P がある程度長ければ，S から作られた D^R を用いて P をパージングすることでコアを決定できる．

文字列を列挙して検索する際には，構文木中でパタンのコアを列挙していき，各コアの周辺で照合を行う．すなわち $P=X_1X_2\ldots X_k$ といった変数列において，そのコアを X_t すると構文木中での X_t にラベル付けされたノード v を列挙した上で，各々について v に隣接する部分木の中に他の $X_1X_2\ldots X_k$ が

存在するかを確認する．$k = O(\log |P|)$ であることから v から $O(\log |P|)$ 個の辺を巡回すればよい．

3.1 構文木で文字列検索を実現するデータ構造

$X \to AB$ という生成規則しか持たないチョムスキー標準形を [7] の圧縮アルゴリズムによって S から直接得られる．これによる構文木 T のノード数は $2u-1$ 個となる．T の各部分木について最左導出によりまとめた木を PT とする．PT は n 個の内部ノードがそれぞれ変数に対応するノード数 $2n+1$ 個の全二分木である．コアに対応するノードを列挙する時にはまとめられた部分木を復元する．PT において同一コアのうち自分の次に右側にくるノードを示すポインタ $lmost\_occ$ と，同一コアの内で最左のものを示すポインタ $next\_occ$ を導入する．

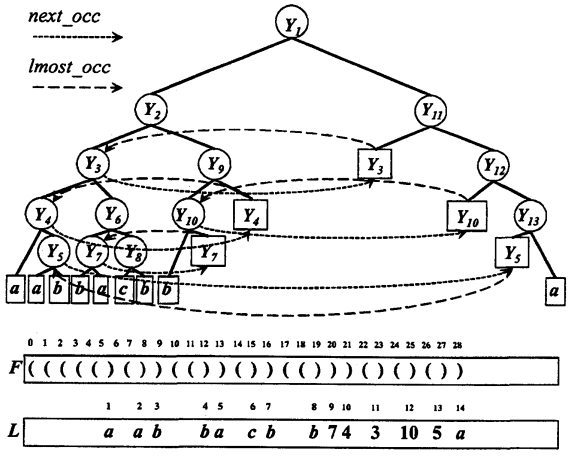

図 1: 木 PT の表現

木構造: PT は前述 2 つのポインタを除くとその形状は全二分木である．よって 2 節の簡潔データ構造を適用できる．そのサイズは $2n + o(n)$ ビットとなる．葉はウェーブレット木で表現する．葉の値を取り出す計算量は $O(\lg n)$ となるが，$lmost\_occ$ と $nextmost\_occ$ に対応できるようになる．括弧列が前置順で並んでいる関係から，構文木生成時の変数とは別に仮想変数 Y を各内部ノードに前置順に配置していく．以下では i をノードの F での位置とする．

- $inner\_rank(i)$: 位置 i の内部ノードに対応する仮想変数
- $inner\_select(k)$: 仮想変数 y_k に対応するノードの F での位置
- $inleaf(i)$: 位置 i の葉ノードが持つラベル
- $refnode(i)$: 位置 i の葉ノードによって参照されているノード

これらの計算から $lmost\_occ(i)$ と $next\_occ(i)$ をいずれも $O(\lg n)$ 時間で求めることができる．

構築時変数と仮想変数の交換のためのデータ構造: パタン P からコアを求めるパーシングはテキスト S から構文木 T を構築する時に割り当てた変数値で行う必要がある．そのためには仮想変数と変数を相互に交換する．このデータ構造は新たにウェーブレット木を導入するなどして $n \lg h + h \lg n + o(n \lg h)$ ビットの領域を用いて $O(\lg h)$ 時間で可能になる．

逆引き辞書: パタン P のパーシングにおいて逆引き辞書が必要になる．このとき仮想変数に置き換えて PT を参照することで記憶するべき対象を，PT においてどちらの子も葉ノードのノードに対応する変数のみとする．これにより $((n+1)/2) \lg n$ ビットの領域において計算量 $O(\lg^2 n)$ で逆引きが可能になる．

3.2 部分復元のためのデータ構造の追加

3.1 節のデータ構造は，長いパタンの検索に効果を発揮するが示されているが $access/locate$ へ対応しなかった．そこで以下のような配列 U を追加する．

配列 U: PT において葉ノードとなっている $n+1$ 個のノードがカバーしているテキスト上の文字列をそれぞれ $s_1, s_2, \ldots, s_{n+1}$ とする．このとき $U = 10^{|s_1|-1}10^{|s_2|-1} \ldots 10^{|s_{n+1}|-1}1$ を定義する．この U は長さ $u+1$ で 1 の数が $n+2$ 個のビット列であり，サイズは $B(u,n) + o(u)$ ビットとすることができる．

この U を用いることで各内部ノードの左分木がテキスト上のどれほどの文字列をカバーしているかを定数時間で求めることもできる．

$access(S,i)$ **操作**:ビット列 U に対する rank/select 操作を利用した access 操作の概要を示す．

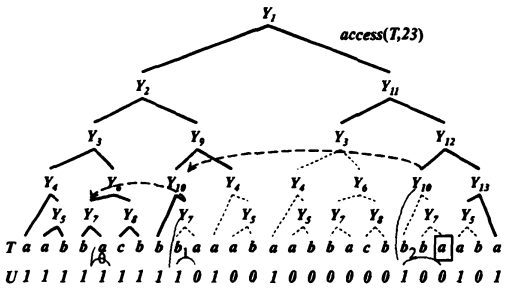

図 2: 配列 U と *access* 操作の概要

1. 以下を構文木での葉，すなわち変数ではなくアルファベットを示す葉に至るまで繰り返す．
2. $S[i]$ の祖先にあたる葉の左からの順番を $k = rank_1(U,i)$ とする．
3. $j = select_1(U,k)$，$s := i - j + 1$
4. k について *lmost_occ* を参照．参照先を根とする部分木の s 文字目を求めるため再帰操作へ．

繰り返しは高々h 回であり，左分木が明示的ではない葉ノードは *lmost_occ* の参照を行うので計算量は $O(h \log n)$ である．$S[i]$ を求めた上で，そこから右側に展開していけば部分文字列復元となる．このとき PT の各葉は配列 L で隣接関係にある．葉は 1 文字以上カバーしているので復元する長さを l とすると，辿る枝の本数は $O(l)$ 本である．計算量は $O((h+l)\lg n)$ となる．

locate(S,P) **操作**: *locate* は *count* に付随して行う．

1. パタン圧縮時にコアがパタン先頭から j 文字離れていることを確認
2. ヒット時にコアが示す部分文字列先頭位置 k
3. $k - j$ を返す．

1. についてはパタン圧縮に並行して行うことができる．2. についても同様に *count* での照合作業に並行して行うことができるので，*locate* の計算量は本来の *count* 操作の計算量を超えることはない．

定理 1 サイズ $1.5n\lg n + B(u,n) + o(n\lg n + u)$ ビットのデータ構造を用いて *access* 操作は $O(h\log n)$ 時間，*locate* 操作にかかる時間は *count* 操作にかかる時間を増やすことはない．

4 おわりに

本稿では，文法圧縮に基づいた索引構造に対して，新たに索引を加えることで検索だけではなく検索位置の特定や部分復元ができることを示した．部分復元の計算量についてはウェーブレット木に変わるデータ構造を利用することで改善できる可能性がある．

参考文献

[1] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, S. S. Rao : " Representing trees of higher degree." *Algorithmica 43(4)*, 275–292, 2005.

[2] R. Grossi, A. Gupta, and J.S. Vitter. "High-order entropy-compressed text indexes." In *SODA04*, pages 636.645, 2004.

[3] J. Jansson, K. Sadakane, and Sung, W.-K.: "Ultra-succinct representation of ordered trees." In *SODA (2007), N. Bansal, K. Pruhs, and C. Stein, Eds., SIAM*, pages. 575–584.

[4] S. Maruyama, H. Sakamoto, M. Baba, H. Ono, K. Sadakane, M. Yamashita. : "Searching Long Patterns from Grammar-Based Compression." Submitting.

[5] J. I. Munro, V. Raman : "Succint Representation of Balanced Parentheses and Static Trees." *SIAM Journal on Computing*, 31(3):762–776, 2001.

[6] R. Raman, V. Raman, S. S. Rao : "Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets." In *Proc. ACM-SIAM SODA*, pages 233–242, 2002.

[7] H. Sakamoto, S. Maruyama, T. Kida and S. Shimozono : "A space-saving approximation algorithms for grammar-based compression." *IEICE Trans. on Information and Systems*, E92-D(2):158–165, 2009.

[8] 馬場雅大, 小野廣隆, 定兼邦彦, 山下雅史. "全二分木の簡潔な表現." 情報処理学会研究報告, Vol.2010-AL-129 No.1, pages 1-8, 2010.